**ПАТТЕРНЫ ПРОГРАММИРОВАНИЯ**

Содержание

[1. Введение в паттерны проектирования 1](#_Toc130404118)

[2. Порождающие паттерны 2](#_Toc130404119)

[2.1 Одиночка 2](#_Toc130404120)

[2.2 Строитель 3](#_Toc130404121)

[3. Структурные паттерны 5](#_Toc130404122)

[3.1 Декоратор 6](#_Toc130404123)

[3.2 Адаптер 8](#_Toc130404124)

[4. Паттерны поведения 10](#_Toc130404125)

[4.1 Стратегия 11](#_Toc130404126)

[4.2 Наблюдатель 13](#_Toc130404127)

[4.3 Команда 16](#_Toc130404128)

[5. Заключение 18](#_Toc130404129)

# **Введение в паттерны проектирования**

Цельное представление о паттернах появилось в 1994 году после выхода нашумевшей книги «Приемы объектно-ориентированного программирования. Паттерны проектирования» от авторов, которых прозвали «Банда четырех».

В книге под паттерном понимается описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте.

Проще говоря, **паттерны проектирования** — это решения распространенных проблем при разработке кода. Также они известны как шаблоны проектирования, паттерны объектно-ориентированного программирования или design patterns. В отличие от готовых функций или библиотек, паттерн представляет собой не конкретный код, а общую концепцию решения проблемы, которую еще нужно подстроить под задачи.

Вы можете спутать паттерны **проектирования** и **архитектурные** паттерны, так как обычно в разговоре разработчики называют и те, и другие просто паттернами. Давайте разберемся, в чем отличия.

Паттерны проектирования предоставляют решения для доработки и построения подсистем на уровне кода. Обычно на их выбор влияет язык программирования, а разработчик сам принимает решение, какой паттерн использовать.

Архитектурные паттерны (архитектурные шаблоны, architectural patterns) рассматриваются на более высоком уровне. Они определяет архитектуру приложения, задают его логику (на какие компоненты/модули будет делиться приложение и каким образом они взаимодействуют) и помогают разработчику понять, как устроен продукт внутри. Выбор паттернов происходит на этапе создания продукта.

Всего существует 23 классических паттерна, которые были описаны в книге «Банды четырех». В зависимости от того, какие задачи решают паттерны, они делятся на три вида — порождающие, структурные и поведенческие.

# **Порождающие паттерны**

Порождающие предназначены для создания экземпляра объекта или группы связанных объектов. К ним относятся:

* Abstract Factory — Абстрактная фабрика
* Builder — Строитель
* Factory Method — Фабричный метод
* Prototype — Прототип
* Singleton — Одиночка

## **Одиночка**

Одиночка (Singleton, Синглтон) - порождающий паттерн, который гарантирует, что для определенного класса будет создан только один объект, а также предоставит к этому объекту точку доступа.

Когда надо использовать Синглтон?

* Когда необходимо, чтобы для класса существовал только один экземпляр

**Синглтон** позволяет создать объект только при его необходимости. Если объект не нужен, то он не будет создан. В этом отличие синглтона от глобальных переменных.

Классическая реализация данного шаблона проектирования на C# выглядит следующим образом:

|  |
| --- |
| class Singleton  {  private static Singleton instance;  private Singleton()  { }  public static Singleton getInstance()  {  if (instance == null)  instance = new Singleton();  return instance;  }  } |

В классе определяется статическая переменная - ссылка на конкретный экземпляр данного объекта и приватный конструктор. В статическом методе getInstance() этот конструктор вызывается для создания объекта, если, конечно, объект отсутствует и равен null.

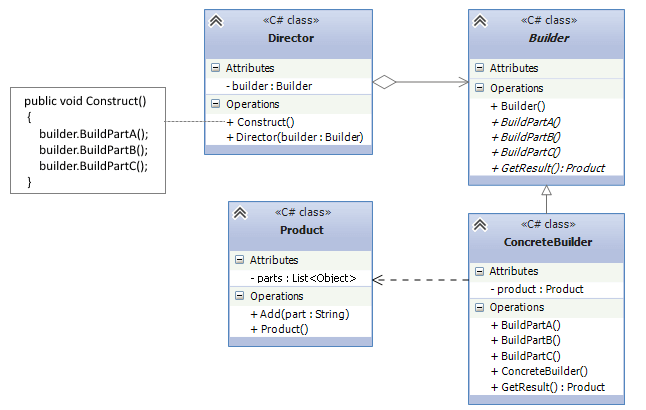
## **Строитель**

Строитель (Builder) - шаблон проектирования, который инкапсулирует создание объекта и позволяет разделить его на различные этапы.

Когда использовать паттерн Строитель?

* Когда процесс создания нового объекта не должен зависеть от того, из каких частей этот объект состоит и как эти части связаны между собой
* Когда необходимо обеспечить получение различных вариаций объекта в процессе его создания

Формально в UML паттерн мог бы выглядеть следующим образом:



Формальное определение на C# могло бы выглядеть так:

|  |
| --- |
| class Client  {  void Main()  {  Builder builder = new ConcreteBuilder();  Director director = new Director(builder);  director.Construct();  Product product = builder.GetResult();  }  } |

|  |
| --- |
| class Director  {  Builder builder;  public Director(Builder builder)  {  this.builder = builder;  }  public void Construct()  {  builder.BuildPartA();  builder.BuildPartB();  builder.BuildPartC();  }  } |

|  |
| --- |
| abstract class Builder  {  public abstract void BuildPartA();  public abstract void BuildPartB();  public abstract void BuildPartC();  public abstract Product GetResult();  } |

|  |
| --- |
| class Product  {  List<object> parts = new List<object>();  public void Add(string part)  {  parts.Add(part);  }  } |

|  |
| --- |
| class ConcreteBuilder : Builder  {  Product product = new Product();  public override void BuildPartA()  {  product.Add("Part A");  }  public override void BuildPartB()  {  product.Add("Part B");  }  public override void BuildPartC()  {  product.Add("Part C");  }  public override Product GetResult()  {  return product;  }  } |

Участники:

* Product: представляет объект, который должен быть создан. В данном случае все части объекта заключены в списке parts.
* Builder: определяет интерфейс для создания различных частей объекта Product
* ConcreteBuilder: конкретная реализация Buildera. Создает объект Product и определяет интерфейс для доступа к нему
* Director: распорядитель - создает объект, используя объекты Builder

# **Структурные паттерны**

Структурные в основном связаны с композицией объектов, с тем, как сущности могут использовать друг друга. К ним относятся:

* Adapter — Адаптер
* Bridge — Мост
* Composite — Компоновщик
* Decorator — Декоратор
* Facade — Фасад
* Flyweight — Приспособленец
* Proxy — Заместитель

## **Декоратор**

[Согласно Википедии](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BE%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)), Decorator — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту.

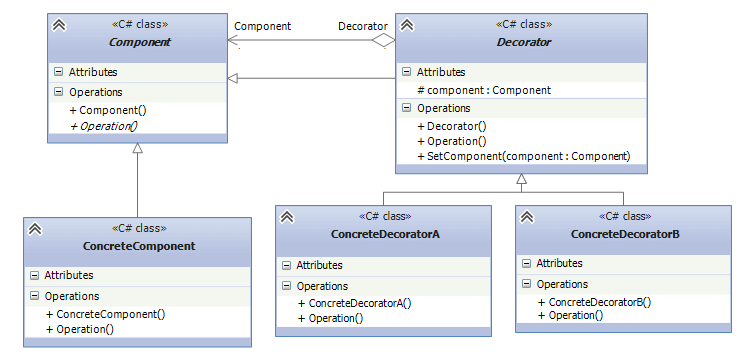
**Проще говоря,** паттерн позволяет добавлять объектам новые функции с помощью обертки без создания отдельного класса.

Для определения нового функционала в классах нередко используется наследование. Декораторы же предоставляет наследованию более гибкую альтернативу, поскольку позволяют динамически в процессе выполнения определять новые возможности у объектов.

Когда следует использовать декораторы?

* + - Когда надо динамически добавлять к объекту новые функциональные возможности. При этом данные возможности могут быть сняты с объекта
    - Когда применение наследования неприемлемо. Например, если нам надо определить множество различных функциональностей и для каждой функциональности наследовать отдельный класс, то структура классов может очень сильно разрастись. Еще больше она может разрастись, если нам необходимо создать классы, реализующие все возможные сочетания добавляемых функциональностей.

Схематически шаблон "Декоратор" можно выразить следующим образом:



Формальная организация паттерна в C# могла бы выглядеть следующим образом:

|  |
| --- |
| abstract class Component  {  public abstract void Operation();  } |

|  |
| --- |
| class ConcreteComponent : Component  {  public override void Operation()  { }  } |

|  |
| --- |
| abstract class Decorator : Component  {  protected Component component;  public void SetComponent(Component component)  {  this.component = component;  }  public override void Operation()  {  if (component != null)  component.Operation();  }  } |

|  |
| --- |
| class ConcreteDecoratorA : Decorator  {  public override void Operation()  {  base.Operation();  }  } |

|  |
| --- |
| class ConcreteDecoratorB : Decorator  {  public override void Operation()  {  base.Operation();  }  } |

Участники:

* Component: абстрактный класс, который определяет интерфейс для наследуемых объектов
* ConcreteComponent: конкретная реализация компонента, в которую с помощью декоратора добавляется новая функциональность
* Decorator: собственно декоратор, реализуется в виде абстрактного класса и имеет тот же базовый класс, что и декорируемые объекты. Поэтому базовый класс Component должен быть по возможности легким и определять только базовый интерфейс.
* Класс декоратора также хранит ссылку на декорируемый объект в виде объекта базового класса Component и реализует связь с базовым классом как через наследование, так и через отношение агрегации.
* Классы ConcreteDecoratorA и ConcreteDecoratorB представляют дополнительные функциональности, которыми должен быть расширен объект ConcreteComponent.

## **Адаптер**

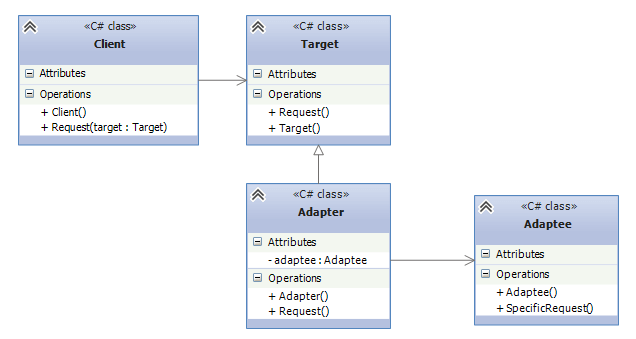
Согласно Википедии, Adapter — структурный шаблон проектирования, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс.

Проще говоря, Adapter позволяет объектам с несовместимыми интерфейсами работать вместе.

Когда надо использовать Адаптер?

* + - Когда необходимо использовать имеющийся класс, но его интерфейс не соответствует потребностям
    - Когда надо использовать уже существующий класс совместно с другими классами, интерфейсы которых не совместимы

Формальное определение паттерна на UML выглядит следующим образом:



Формальное описание адаптера объектов на C# выглядит таким образом:

|  |
| --- |
| class Client  {  public void Request(Target target)  {  target.Request();  }  } |

|  |
| --- |
| class Target  {  public virtual void Request()  { }  } |

|  |
| --- |
| class Adapter : Target  {  private Adaptee adaptee = new Adaptee();  public override void Request()  {  adaptee.SpecificRequest();  }  } |

|  |
| --- |
| class Adaptee  {  public void SpecificRequest()  { }  } |

Участники:

* + - Target: представляет объекты, которые используются клиентом
    - Client: использует объекты Target для реализации своих задач
    - Adaptee: представляет адаптируемый класс, который мы хотели бы использовать у клиента вместо объектов Target
    - Adapter: собственно адаптер, который позволяет работать с объектами Adaptee как с объектами Target.

То есть клиент ничего не знает об Adaptee, он знает и использует только объекты Target. И благодаря адаптеру мы можем на клиенте использовать объекты Adaptee как Target

# **Паттерны поведения**

Поведенческие связаны с распределением обязанностей между объектами. Их отличие от структурных шаблонов заключается в том, что они описывают не только структуру, но и способы общения между ними. К ним относятся:

* Chain of responsibility — Цепочка обязанностей
* Command — Команда
* Interpreter — Интерпретатор
* Iterator — Итератор
* Mediator — Посредник
* Memento — Хранитель
* Observer — Наблюдатель
* State — Состояние
* Strategy — Стратегия
* Template method — Шаблонный метод
* Visitor — Посетитель

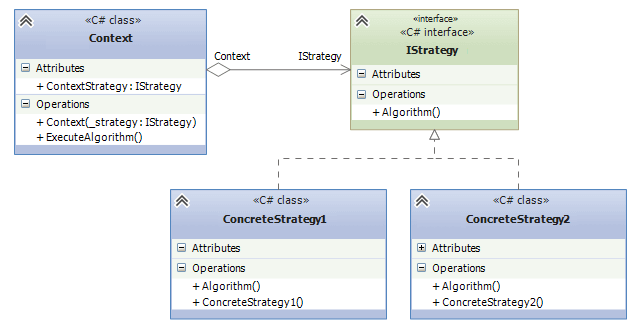
## **Стратегия**

Паттерн Стратегия (Strategy) представляет шаблон проектирования, который определяет набор алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость. В зависимости от ситуации мы можем легко заменить один используемый алгоритм другим. При этом замена алгоритма происходит независимо от объекта, который использует данный алгоритм.

Когда использовать стратегию?

* + - Когда есть несколько родственных классов, которые отличаются поведением. Можно задать один основной класс, а разные варианты поведения вынести в отдельные классы и при необходимости их применять
    - Когда необходимо обеспечить выбор из нескольких вариантов алгоритмов, которые можно легко менять в зависимости от условий
    - Когда необходимо менять поведение объектов на стадии выполнения программы
    - Когда класс, применяющий определенную функциональность, ничего не должен знать о ее реализации

Формально паттерн Стратегия можно выразить следующей схемой UML:



Формальное определение паттерна на языке C# может выглядеть следующим образом:

|  |
| --- |
| public interface IStrategy  {  void Algorithm();  } |

|  |
| --- |
| public class ConcreteStrategy1 : IStrategy  {  public void Algorithm()  { }  } |

|  |
| --- |
| public class ConcreteStrategy2 : IStrategy  {  public void Algorithm()  { }  } |

|  |
| --- |
| public class Context  {  public IStrategy ContextStrategy { get; set; }  public Context(IStrategy \_strategy)  {  ContextStrategy = \_strategy;  }  public void ExecuteAlgorithm()  {  ContextStrategy.Algorithm();  }  } |

Участники:

* + - Интерфейс IStrategy, который определяет метод Algorithm(). Это общий интерфейс для всех реализующих его алгоритмов. Вместо интерфейса здесь также можно было бы использовать абстрактный класс.
    - Классы ConcreteStrategy1 и ConcreteStrategy, которые реализуют интерфейс IStrategy, предоставляя свою версию метода Algorithm(). Подобных классов-реализаций может быть множество.
    - Класс Context хранит ссылку на объект IStrategy и связан с интерфейсом IStrategy отношением агрегации.

В данном случае объект IStrategy заключена в свойстве ContextStrategy, хотя также для нее можно было бы определить приватную переменную, а для динамической установки использовать специальный метод.

## **Наблюдатель**

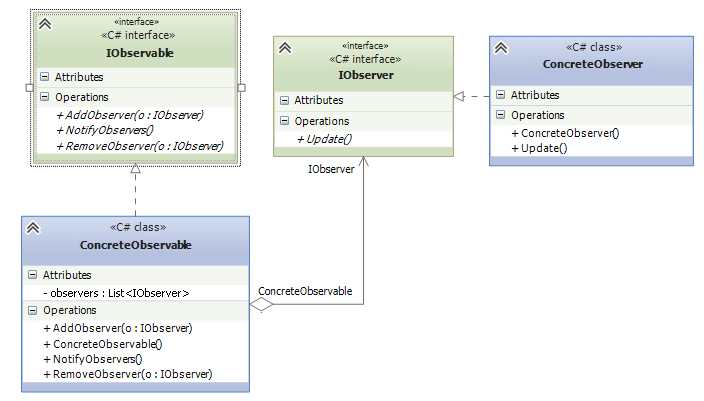
Паттерн "Наблюдатель" (Observer) представляет поведенческий шаблон проектирования, который использует отношение "один ко многим". В этом отношении есть один наблюдаемый объект и множество наблюдателей. И при изменении наблюдаемого объекта автоматически происходит оповещение всех наблюдателей.

Данный паттерн еще называют Publisher-Subscriber (издатель-подписчик), поскольку отношения издателя и подписчиков характеризуют действие данного паттерна: подписчики подписываются email-рассылку определенного сайта. Сайт-издатель с помощью email-рассылки уведомляет всех подписчиков о изменениях. А подписчики получают изменения и производят определенные действия: могут зайти на сайт, могут проигнорировать уведомления и т.д.

Когда использовать паттерн Наблюдатель?

* + - Когда система состоит из множества классов, объекты которых должны находиться в согласованных состояниях
    - Когда общая схема взаимодействия объектов предполагает две стороны: одна рассылает сообщения и является главным, другая получает сообщения и реагирует на них. Отделение логики обеих сторон позволяет их рассматривать независимо и использовать отдельно друга от друга.
    - Когда существует один объект, рассылающий сообщения, и множество подписчиков, которые получают сообщения. При этом точное число подписчиков заранее неизвестно и процессе работы программы может изменяться.

С помощью диаграмм UML данный шаблон можно выразить следующим образом:



Формальное определение паттерна на языке C# может выглядеть следующим образом:

|  |
| --- |
| interface IObservable  {  void AddObserver(IObserver o);  void RemoveObserver(IObserver o);  void NotifyObservers();  } |

|  |
| --- |
| class ConcreteObservable : IObservable  {  private List<IObserver> observers;  public ConcreteObservable()  {  observers = new List<IObserver>();  }  public void AddObserver(IObserver o)  {  observers.Add(o);  }  public void RemoveObserver(IObserver o)  {  observers.Remove(o);  }  public void NotifyObservers()  {  foreach (IObserver observer in observers)  observer.Update();  }  } |

|  |
| --- |
| interface IObserver  {  void Update();  } |

|  |
| --- |
| class ConcreteObserver : IObserver  {  public void Update()  {  }  } |

Участники

* + - IObservable: представляет наблюдаемый объект. Определяет три метода: AddObserver() (для добавления наблюдателя), RemoveObserver() (удаление набюдателя) и NotifyObservers() (уведомление наблюдателей)
    - ConcreteObservable: конкретная реализация интерфейса IObservable. Определяет коллекцию объектов наблюдателей.
    - IObserver: представляет наблюдателя, который подписывается на все уведомления наблюдаемого объекта. Определяет метод Update(), который вызывается наблюдаемым объектом для уведомления наблюдателя.
    - ConcreteObserver: конкретная реализация интерфейса IObserver.

При этом наблюдаемому объекту не надо ничего знать о наблюдателе кроме того, что тот реализует метод Update(). С помощью отношения агрегации реализуется слабосвязанность обоих компонентов. Изменения в наблюдаемом объекте не виляют на наблюдателя и наоборот.

В определенный момент наблюдатель может прекратить наблюдение. И после этого оба объекта - наблюдатель и наблюдаемый могут продолжать существовать в системе независимо друг от друга.

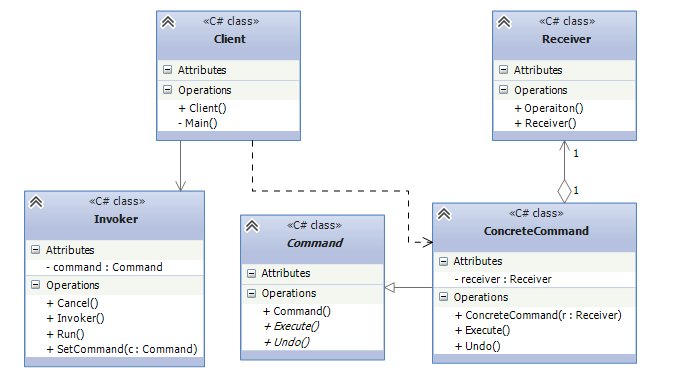
## **Команда**

Команда (англ. Command) — поведенческий шаблон проектирования, используемый при объектно-ориентированном программировании, представляющий действие. Объект команды заключает в себе само действие и его параметры.

Паттерн "Команда" (Command) позволяет инкапсулировать запрос на выполнение определенного действия в виде отдельного объекта. Этот объект запроса на действие и называется командой. При этом объекты, инициирующие запросы на выполнение действия, отделяются от объектов, которые выполняют это действие.

Команды могут использовать параметры, которые передают ассоциированную с командой информацию. Кроме того, команды могут ставиться в очередь и также могут быть отменены.

Схематично в UML паттерн Команда представляется следующим образом:



Формальное определение на языке C# может выглядеть следующим образом:

|  |
| --- |
| abstract class Command  {  public abstract void Execute();  public abstract void Undo();  } |

|  |
| --- |
| class ConcreteCommand : Command  {  Receiver receiver;  public ConcreteCommand(Receiver r)  {  receiver = r;  }  public override void Execute()  {  receiver.Operation();  }  public override void Undo()  { }  } |

|  |
| --- |
| class Receiver  {  public void Operation()  { }  } |

|  |
| --- |
| class Invoker  {  Command command;  public void SetCommand(Command c)  {  command = c;  }  public void Run()  {  command.Execute();  }  public void Cancel()  {  command.Undo();  }  } |

Участники

* Command: интерфейс, представляющий команду. Обычно определяет метод Execute() для выполнения действия, а также нередко включает метод Undo(), реализация которого должна заключаться в отмене действия команды
* ConcreteCommand: конкретная реализация команды, реализует метод Execute(), в котором вызывается определенный метод, определенный в классе Receiver
* Receiver: получатель команды. Определяет действия, которые должны выполняться в результате запроса.
* Invoker: инициатор команды - вызывает команду для выполнения определенного запроса
* Client: клиент - создает команду и устанавливает ее получателя с помощью метода SetCommand()

Таким образом, инициатор, отправляющий запрос, ничего не знает о получателе, который и будет выполнять команду. Кроме того, если нам потребуется применить какие-то новые команды, мы можем просто унаследовать классы от абстрактного класса Command и реализовать его методы Execute и Undo.

# **Заключение**

Паттерны проектирования — это решения распространенных проблем при разработке кода. Их знание и использование позволяет экономить время, используя готовые решения, стандартизировать код и повысить общий программистский словарь.

Основное отличие паттернов проектирования от архитектурных в том, что первые предоставляют решения на уровне кода. Обычно на их выбор влияет язык программирования, а разработчик сам принимает решение, какой паттерн использовать. Архитектурные паттерны определяют архитектуру приложения, задают его логику и помогают разработчику понять, как устроен продукт внутри.

В зависимости от того, какие задачи решают паттерны проектирования, они делятся на три вида: порождающие, структурные и поведенческие.